$1.50

Apple Assembly Line

Volume 3 -- Issue 6 March, 1983

In This Issue...

S-C Macro Assembler Version 1.1

That's right, Version 1.1! I've added all the most-requested new features, corrected those few lingering problems, and it's almost ready. Look inside for more details.

A New Screen-Oriented Editor

Several people have asked about a screen-oriented editor for the S-C Macro Assembler. Well, Mike Laumer has come up with one for you. It runs with the Language Card version of the Macro Assembler, in the unused bank. I still prefer a line editor, but Bill is rapidly falling in love with the new screen editor. Now everyone has a choice! See Mike's ad inside.

65C02

Many of you have expressed an interest in the new Rockwell R65C02 microprocessor. Well, I still haven't heard any more than I mentioned a couple of months ago. We're as eager as you are to get a sample. We'll have a detailed report as soon as we know more.

All About PTRGET & GETARYPT.....Bob Sander-Cederlof

Both Leo Reich and E. Melioli have asked for some clarification on how to pass array variables between Applesoft programs and assembly language programs. I hope this little article will be of some help to them.

The Variable Tables:

We need to start with a look at the structure of the Applesoft variable tables. There are two variable tables: one for simple variables, and the other for arrays. (You might turn to page 137 of the Applesoft Reference Manual now.) Entries in these tables include the variable names; some codes to distinguish real, integer, and string variables; and the value if numeric. String variables include the length of the string and the address of the string, but not the string itself.

The address of the start of the simple variable table is kept in $69,$6A. The next pair, $6B and $6C, hold the address of the end of the simple variable table plus one. This happens to also be the address of the beginning of the array variable table. The address of the end of the arrays plus one is kept in $6D,$6E. The actual string values may be inside the program itself, in the case of "string" values; or in the space between the top of the array variable table and HIMEM.

Here is a picture, with a few more pointers thrown in for good measure:

```
(73.74) -->    HIMEM

               <string values>

(6F.70) -->    String Bottom

               <free space>

(6D.6E) -->    Free Memory Bottom

               <arrays>

(6B.6C) -->    Array Variable Bottom

               <variables>

(69.6A) -->    Simple Variable Bottom

               <program>

(67.68) -->    Program Bottom
```

Inside an Array:

Let's look a little closer at the array variable space. Each array in there consists of a header part and a data part. The header part contains the name, flags to indicate real-integer-string, the offset to the next array, the number of dimensions,

S-C Macro Assembler (the best there is!).....$80.00
Upgrade from Version 4.0 to MACRO.....$27.50
Source code of Version 4.0 on disk.....$95.00
Fully commented, easy to understand and modify to your own tastes.
S-C Macro Assembler /// $100.00
Preliminary version. Call or write for details.

S-C Word Processor.....$50.00
As is, with fully commented source code. Needs S-C Macro Assembler.

Applesoft Source Code on Disk.....$50.00
Very heavily commented. Requires Applesoft and S-C Assembler.

ES-CAPE: Extended S-C Applesoft Program Editor.....$60.00

AAL Quarterly Disks.....each $15.00
Each disk contains all the source code from three issues of "Apple Assembly Line", to save you lots of typing and testing time.
QD#1: Oct-Dec 1980 QD#2: Jan-Mar 1981 QD#3: Apr-Jun 1981
QD#4: Jul-Sep 1981 QD#5: Oct-Dec 1981 QD#6: Jan-Mar 1982
QD#7: Apr-Jun 1982 QD#8: Jul-Sep 1982 QD#9: Oct-Dec 1982
QD#10: Jan-Mar 1983

Double Precision Floating Point for Applesoft.....$50.00
Provides 21-digit precision for Applesoft programs.
Includes sample Applesoft subroutines for standard math functions.

FLASH! Integer BASIC Compiler (Laumer Research)..... $79.00
Source Code for FLASH! Runtime Package.....$39.00

Super Disk Copy III (Sensible Software).....(reg. $30.00) $27.00
Amper-Magic (Anthro-Digital).....(reg. $75.00) $67.50
Amper-Magic Volume 2 (Anthro-Digital).....(reg. $35.00) $30.00
Quick-Trace (Anthro-Digital).....(reg. $50.00) $45.00
Cross-Reference and Dis-Assembler (Rak-Ware).....$45.00
The Incredible JACK!.....(reg. $129.00) $99.00

Blank Diskettes (with hub rings).....package of 20 for $50.00
Small 3-ring binder with 10 vinyl disk pages and disks.....$36.00
Vinyl disk pages, 6"x8.5", hold one disk each.....10 for $6.00
Reload your own NEC PC-8023 ribbon cartridges.....each ribbon $5.00
Reload your own NEC Spinwriter Multi-Strike Film cartridges....each $2.50
Diskette Mailing Protectors.....10-99: 40 cents each
100 or more: 25 cents each

ZIF Game Socket Extender.....$20.00
Ashby Shift-Key Mod.....$15.00
Lower-Case Display Encoder ROM.....$25.00
Only Revision level 7 or later Apples.

Books, Books, Books.....compare our discount prices!
"Enhancing Your Apple II, vol. 1", Lancaster.....($15.95) $15.00
"Incredible Secret Money Machine", Lancaster.....($7.95) $7.50
"Micro Cookbook, vol. 1", Lancaster.....($15.95) $15.00
"Beneath Apple DOS", Worth & Lechner.....($19.95) $18.00
"Bag of Tricks", Worth & Lechner, with diskette.....($39.95) $36.00
"Apple Graphics & Arcade Game Design", Stanton.....($19.95) $18.00
"Assembly Lines: The Book", Roger Wagner.....($19.95) $18.00
"What's Where in the Apple", Second Edition.....($24.95) $23.00
"What's Where Guide" (updates first edition).....($9.95) $9.00
"6502 Assembly Language Programming", Leventhal.....($16.99) $16.00
"6502 Subroutines", Leventhal.....($12.99) $12.00
"MICRO on the Apple--1", includes diskette.....($24.95) $23.00
"MICRO on the Apple--2", includes diskette.....($24.95) $23.00
"MICRO on the Apple--3", includes diskette.....($24.95) $23.00

Add $1 per book for US postage. Foreign orders add postage needed.

*** S-C SOFTWARE, P. O. BOX 280300, Dallas, TX 75228 ***
*** (214) 324-2050 ***
*** We take Master Charge, VISA and American Express ***

and each dimension. The data part contains all the numeric values for real or integer arrays, and all the string length-address pairs for string arrays.

Here is a picture of the header part:

Bytes	Contents
0,1	Name of Array
2,3	Offset
4	# of dimensions
5,6	last dimension
...	
x,y	first dimension

The sign bits in each byte of the name combine to tell what type of array variable this is. If both bytes are positive, it is a real array; if both are negative, it is integer. Contrary to what it says on page 137 of the Applesoft manual, if the 1st byte is positive and the 2nd byte negative it is a string array. The manual has it backwards.

The value in the offset can be added to the address of the first byte of the header to give the address of the first byte of the header of the next array (or the end of arrays if there are no more).

The number of dimensions is one byte, which obviously means no more than 255 dimensions per array. Oh well! In my sample below I assume that no more than 120 dimensions have been declared. If you try to declare more than that, you will see how hard it is.

The dimensions are stored in backward order, last dimension first. Why? Why not? It has to do with the order they are used in calculating position for an individual element. Each dimension is also one larger than you declare in the DIM statement, because subscripts start at 0.

The data part of an array consists of the elements ordered so that the first subscript changes fastest. That is, element X(2,10) directly follows element X(1,10) in memory. Integer array elements are two bytes each, with the high byte first. Note: this is just about the only place in all the 6502 kingdom where you will find highbytes first on 16-bit values!

Real array elements take five bytes each: one byte for exponent, and four for mantissa. String array elements take three bytes each: one for length of the string, and two for the address of the string. Note: the string array elements DO NOT hold the string data, but only the address and length of that data!

Getting to the Point:

There is a powerful and much-used subroutine in the Applesoft ROMs which will find a particular variable in the tables. It is called PTRGET, and starts at $DFE3. It is too complicated

to fully explain here, but here is what it does:

1. Reads the variable name from the program text.
2. Determines whether the variable is a simple one or an array.
3. Searches the appropriate table for the name.
4. If the name is not found, create a variable of the approriate type (simple or array; integer, real, or string).
5. Return with the address of the variable in Y,A (high-byte in the Y-register, low-byte in the A-register) and also in $83,84.

That is usually what happens. Actually there are several different entry points and two control bytes which modify PTRGET's behavior depending on the caller's whims. DIMFLG ($10) is set non-zero when called by the DIM-statement processor, and is otherwise cleared to zero. SUBFLG ($14) has four different states:

$00 -- normal value
$40 -- when called by GTARYPT
$80 -- when called to process "DEF FN"
$C1-$DA -- when called to process "FN"

We are concerned with the two cases SUBFLG = 0 and SUBFLG = $40, with DIMFLG = 0. Since the point of this whole article is to clarify access to array variables, I will concentrate on the main entry at $DFE3 and the GETARYPT subroutine at $F7D9. $DFE3 sets SUBFLG = 0, while GETARYPT sets SUBFLG = $40.

When we want to find an individual element inside an array, we call PTRGET at $DFE3. When we want to find the whole array, we call GETARYPT at $F7D9. GETARYPT is used by the STORE and RECALL Applesoft statements (which you might not realize even exist, since their function is only of interest to cassette tape users!)

The "& X" calls in the following program use PTRGET to find an array element.

On the other hand, if we want to sort the array, or if we want to save it all on disk, or some other feat which requires seeing the whole thing at once, we need to call GETARYPT. Then we can even find out how many subscripts were used in the DIM statement, and what the value of each dimension is. GETARYPT returns with the starting address of the whole array in $9B and $9C (called LOWTR).

The "& Y" call in the program prints out the starting address and length of each string of a string array.

I hope that as you work through the descriptions and examples above they are of some help.

QUICKTRACE

relocatable program traces and displays the actual machine operations, *while* it is running without interfering with those operations. Look at these *FEATURES:*

***Single-Step** mode displays the last instruction, next instruction, registers, flags, stack contents, and six user-definable memory locations.*

***Trace** mode gives a running display of the Single-Step information and can be made to stop upon encountering any of nine user-definable conditions.*

***Background** mode permits tracing with no display until it is desired. Debugged routines run at near normal speed until one of the stopping conditions is met, which causes the program to return to Single-Step.*

***QUICKTRACE** allows changes to the stack, registers, stopping conditions, addresses to be displayed, and output destinations for all this information. All this can be done in Single-Step mode while running.*

***Two optional display formats** can show a sequence of operations at once. Usually, the information is given in four lines at the bottom of the screen.*

***QUICKTRACE** is completely transparent to the program being traced. It will not interfere with the stack, program, or I/O.*

***QUICKTRACE** is relocatable to any free part of memory. Its output can be sent to any slot or to the screen.*

***QUICKTRACE** is completely compatible with programs using Applesoft and Integer BASICs, graphics, and DOS. (Time dependent DOS operations can be bypassed.) It will display the graphics on the screen while **QUICKTRACE** is alive.*

***QUICKTRACE** is a beautiful way to show the incredibly complex sequence of operations that a computer goes through in executing a program*

See these programs at participating Computerland and other fine computer stores.

Anthro - Digital Software, Inc.
P.O. Box 1385 Pittsfield, MA 01202

QUICKTRACE *$50*

```
  100  DIM A$(7,9)
  110 A$(3,5) = "ABCDEFG":A$(2,3)   "MNOPQRST"
  120  & X,A$(3,5)
  140  & X,A$(2,3)
  200  REM
  210  FOR J = 0 TO 7: FOR K   0 TO 9
  215 A$ = ""
  220  FOR I = 1 TO  RND (1) * 5 + 5
  230 A$ = A$ +  CHR$ ( RND (1) * 26 + 65)
  240  NEXT I
  245  PRINT J"  "K"  "A$
  250 A$(J,K) = A$: NEXT K: NEXT J
  260  & Y,A$
```

```
                    1000 *  S.ARRAYS
                    1010 *--------------------------------
00B1-               1020 CHRGET    .EQ $B1
DEBE-               1030 CHKCOM    .EQ $DEBE
DEC0-               1040 SYNCHR    .EQ $DEC0
DFE3-               1050 PTRGET    .EQ $DFE3
F7D9-               1060 GETARYPT  .EQ $F7D9
F941-               1070 PRNTAX    .EQ $F941
FD8E-               1080 CROUT     .EQ $FD8E
FDDA-               1090 PRHEX     .EQ $FDDA
FDED-               1100 COUT      .EQ $FDED
                    1110 *--------------------------------
0000-               1120 LENGTH        .EQ 0
0001-               1130 STRING.ADDR   .EQ 1,2
0003-               1140 ELEMENT.PNTR  .EQ 3,4
0005-               1150 ARRAY.END     .EQ 5,6
                    1160 *--------------------------------
                    1170         .OR $300
                    1180
0300- A9 0B         1190 START   LDA #X
0302- 8D F6 03      1200         STA $3F6
0305- A9 03         1210         LDA /X
0307- 8D F7 03      1220         STA $3F7
030A- 60            1230         RTS
                    1240 *--------------------------------
                    1250 *  GET ONE ARRAY ELEMENT
                    1260 *--------------------------------
030B- C9 58         1270 X       CMP #'X
030D- D0 2C         1280         BNE Y
030F- 20 B1 00      1290         JSR CHRGET
0312- 20 BE DE      1300         JSR CHKCOM       BE SURE COMMA IS NEXT
0315- 20 E3 DF      1310         JSR PTRGET
                    1320 *--------------------------------
                    1330 *   NOW $83,84 POINTS AT A$(3,5)
                    1340 *--------------------------------
0318- A0 00         1350         LDY #0           FIRST BYTE IS STRING LENGTH
031A- B1 83         1360         LDA ($83),Y      GET LENGTH
031C- 85 00         1370         STA LENGTH
031E- C8            1380         INY              NEXT TWO BYTES POINT
031F- B1 83         1390         LDA ($83),Y      AT STRING VALUE
0321- 85 01         1400         STA STRING.ADDR
0323- C8            1410         INY
0324- B1 83         1420         LDA ($83),Y
0326- 85 02         1430         STA STRING.ADDR+1
                    1440 *--------------------------------
                    1450 *   NOW LET'S PRINT THE STRING, JUST FOR FUN
                    1460 *--------------------------------
0328- A0 00         1470         LDY #0
032A- C4 00         1480 .1      CPY LENGTH
032C- B0 0A         1490         BCS .2           FINISHED
032E- B1 01         1500         LDA (STRING.ADDR),Y
0330- 09 80         1510         ORA #$80
0332- 20 ED FD      1520         JSR COUT
0335- C8            1530         INY
0336- D0 F2         1540         BNE .1           ...ALWAYS
0338- 4C 8E FD      1550 .2      JMP CROUT
```

```
                     1560 *--------------------------------
                     1570 *  GET ENTIRE ARRAY
                     1580 *--------------------------------
033B- A9 59          1590 Y' LDA #'Y
033D- 20 C0 DE       1600    JSR SYNCHR
0340- 20 BE DE       1610    JSR CHKCOM
0343- 20 D9 F7       1620    JSR GETARYPT
                     1630 *--------------------------------
                     1640 *   NOW $9B,9C HAVE ADDRESS OF START OF ARRAY
                     1650 *   NEED TO MOVE POINTER UP TO FIRST ELEMENT
                     1660 *--------------------------------
0346- A0 04          1670    LDY #4       POINT AT LSB OF # DIMENSIONS
0348- B1 9B          1680    LDA ($9B),Y
034A- 0A             1690    ASL          DOUBLE IT (IGNORE MSB, #<120)
034B- 69 05          1700    ADC #5       POINT AT FIRST ELEMENT
034D- 85 9D          1710    STA $9D
034F- A0 02          1720    LDY #2       POINT AT LSB OF OFFSET
0351- 18             1730    CLC          COMPUTE ADDRESS JUST PAST END
0352- A5 9B          1740    LDA $9B                          OF ARRAY
0354- 71 9B          1750    ADC ($9B),Y
0356- 85 05          1760    STA ARRAY.END
0358- A5 9C          1770    LDA $9C      MSB
035A- C8             1780    INY
035B- 71 9B          1790    ADC ($9B),Y
035D- 85 06          1800    STA ARRAY.END+1
                     1810 *--------------------------------
                     1820 *   NOW COMPUTE FULL ADDRESS OF FIRST ELEMENT
                     1830 *--------------------------------
035F- 18             1840    CLC
0360- A5 9D          1850    LDA $9D
0362- 65 9B          1860    ADC $9B
0364- 85 03          1870    STA ELEMENT.PNTR
0366- A5 9C          1880    LDA $9C
0368- 69 00          1890    ADC #0
036A- 85 04          1900    STA ELEMENT.PNTR+1
```

```
                     1910 *-------------------------------
                     1920 *       NOW WALK THROUGH STRINGS
                     1930 *-------------------------------  ELEMENT
036C- A0 00          1940 .1      LDY #0                    POINT AT FIRST
036E- B1 03          1950         LDA (ELEMENT.PNTR),Y      GET LENGTH
0370- 85 00          1960         STA LENGTH
0372- C8             1970         INY
0373- B1 03          1980         LDA (ELEMENT.PNTR),Y      GET ADDRESS
0375- AA             1990         TAX
0376- C8             2000         INY
0377- B1 03          2010         LDA (ELEMENT.PNTR),Y
0379- 20 41 F9       2020         JSR PRNTAX
037C- A9 BA          2030         LDA #':+$80
037E- 20 ED FD       2040         JSR $FDED
0381- A9 A0          2050         LDA #' +$80
0383- 20 ED FD       2060         JSR $FDED
0386- 20 ED FD       2070         JSR $FDED
0389- A5 00          2080         LDA LENGTH
038B- 20 DA FD       2090         JSR PRHEX
038E- 20 8E FD       2100         JSR CROUT
                     2110 *-------------------------------
0391- 18             2120         CLC
0392- A9 03          2130         LDA #3
0394- 65 03          2140         ADC ELEMENT.PNTR
0396- 85 03          2150         STA ELEMENT.PNTR
0398- A5 04          2160         LDA ELEMENT.PNTR+1
039A- 69 00          2170         ADC #0
039C- 85 04          2180         STA ELEMENT.PNTR+1
                     2190 *-------------------------------
039E- A5 03          2200         LDA ELEMENT.PNTR
03A0- C5 05          2210         CMP ARRAY.END
03A2- A5 04          2220         LDA ELEMENT.PNTR+1
03A4- E5 06          2230         SBC ARRAY.END+1
03A6- 90 C4          2240         BCC .1
03A8- 60             2250         RTS
```

Macro Can Build Macros.....Mike Laumer

The S-C Macro Assembler can do a lot of things even its designer never dreamed of. The macro capability may be limited compared to mainframe systems, but it still has a lot of power.

A few days ago I got a bright idea that maybe you could even define macros inside macros, or write a macro that builds new macros. Lo and behold, it works! Here is what I tried:

```
1000          .MA BLD
1010          ]1
1020          ]2
1030          ]3
1040          ]4
1050          .EM
```

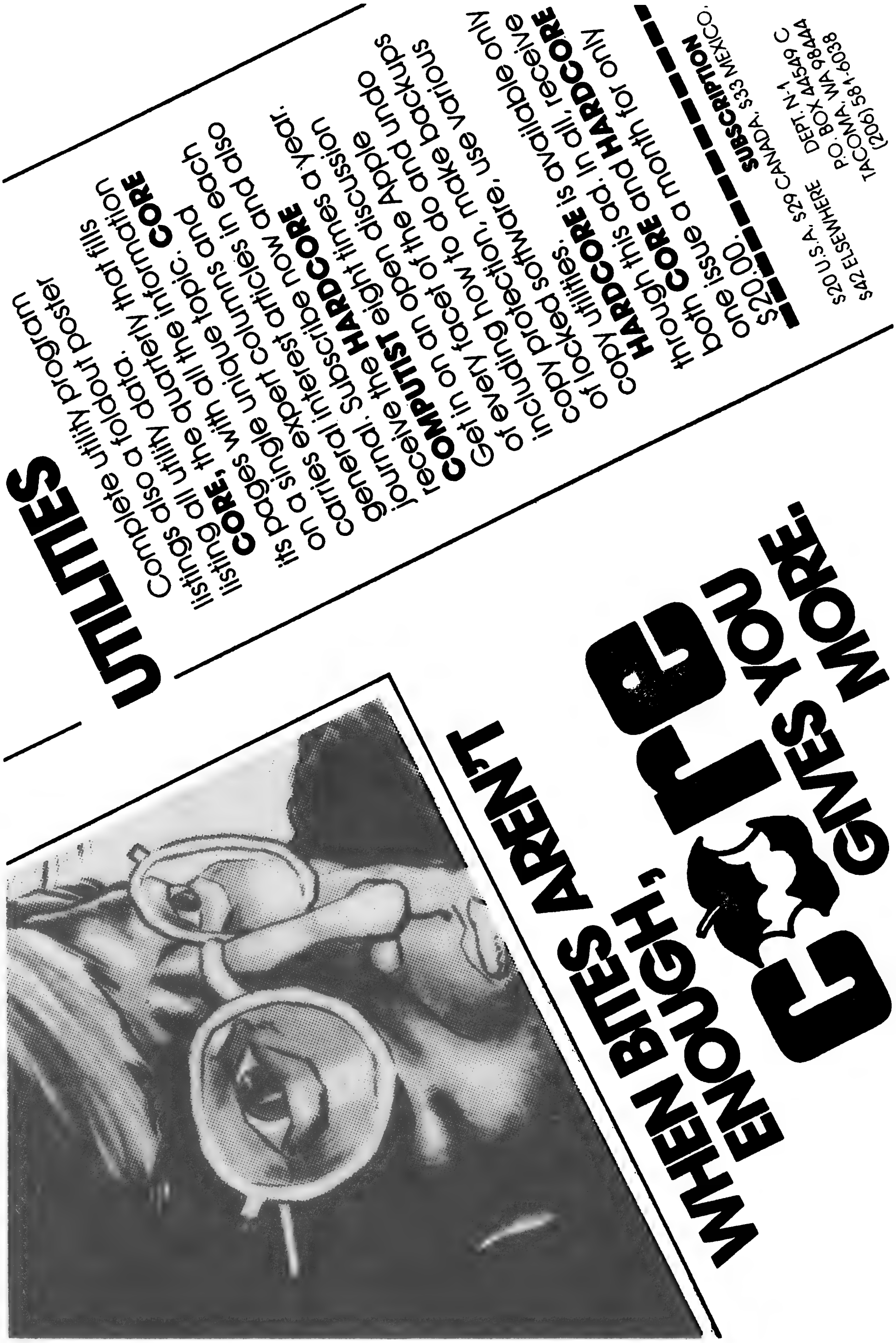
Notice that every line from the opcode field on is defined by a macro parameter. I called it with lines like this:

```
1060          >BLD ".MA ATOB","LDA A","STA B",".EM"
1070          >BLD ".MA BTOA","LDA B","STA A",".EM"
```

Here is how it all looks when you type ASM:

```
                  1010          .MA BLD
                  1020          ]1
                  1030          ]2
                  1040          ]3
                  1050          ]4
                  1060          .EM
0800-             1070          >BLD ".MA ATOB","LDA A","STA B",".EM"
                  0000>          .MA ATOB
                  0000>          LDA A
                  0000>          STA B
                  0000>          .EM
0800-             1080          >BLD ".MA BTOA","LDA B","STA A",".EM"
                  0000>          .MA BTOA
                  0000>          LDA B
                  0000>          STA A
                  0000>          .EM
                  1090 *--------------------------------
0800-             1100 A        .BS 1
0801-             1110 B        .BS 1
                  1120 *--------------------------------
0802-             1130          >ATOB
0802- AD 00 08    0000>          LDA A
0805- 8D 01 08    0000>          STA B
0808-             1140          >BTOA
0808- AD 01 08    0000>          LDA B
080B- 8D 00 08    0000>          STA A
```

I don't know whether this is really useful or not.... If you think of a way to use it that is significant, I'd like to hear from you!



WHEN BITES AREN'T ENOUGH, CORE GIVES YOU MORE.
UTILITIES
Complete utility program listings also a foldout poster listing all utility data.
CORE, the quarterly that fills its pages with all the information on a single unique topic. CORE carries expert columns and general interest articles in each journal. Subscribe now and also receive the HARDCORE COMPUTIST eight times a year. Get in on an open discussion of every facet of the Apple including how to do and undo copy protection, make backups of locked software, use various copy utilities.
HARDCORE is available only through this ad. In all, receive both CORE and HARDCORE one issue a month for only $20.00.
SUBSCRIPTION
$20 U.S.A. $29 CANADA, $33 MEXICO, $42 ELSEWHERE
DEPT. N-1
P.O. BOX 44549 C
TACOMA, WA 98444
(206) 581-6038

Epson MX-80 Text Screen Dump.....Ulf Schlichtmann
West Germany

Here is a short machine language program I wrote some time ago when I was working on a data-base program. It permits you to make a hard copy of the Apple text screen. It was written for an Epson MX-80 with Epson's Apple II Interface kit type 2, but with just one slight modification it should work with any other printer or interface as well.

I thought readers of AAL might have a use for this, especially after seeing a similar program in NIBBLE (Vol. 3 No. 3 pages 147-148) that was over three times longer to produce exactly the same result! The authors of that program required 149 bytes, and even used self-modifying code. My routine is only 40 bytes long.

There is one difference: in the NIBBLE program KSWL,H is changed so that the routine will be invoked every time control-P is pressed; also the ampersand vector is set up to re-install the KSWL,H vector whenever needed. I don't need these features, but even when they are added my program is still only about 78 bytes long (and WITHOUT any self-modifying code!).

Lines 1180-1200 direct all following output to the printer, and is equivalent to the Applesoft statements:

```
PR#1 : PRINT
```

Next I store $8D (left over from MON.CROUT) as the number of columns for the printer, since any number greater than 40 will disable output to the screen. If you have a different printer interface card, you may need to use a different location than $678+SLOT. It should be stated somewhere in the printer interface manual. This is the slight modification I mentioned earlier.

Then I use the Applesoft VTAB routine to calculate the base address for each line. The entry point I chose requires the X-register to be loaded with the number of the desired line (starting with zero for the top-most line). The base address will then be stored in BASL,H. [Note that using AS.VTAB means that this program will only work if Applesoft is switched on. If you call this when the other memory bank is on, no telling what might happen!]

Next I let Y run from 0 to 39 to pick up all the characters in that particular line via indirect addressing. Each character is immediately fed to the printer. Upon completing a line, I call MON.CROUT to cause the printer to print the line. When I have sent all 24 lines, I then redirect output to the CRT and rehook DOS (lines 1340-1350).

Of course, there are a lot of possibilities for adding features to my basic screen dumper. The next version below does not rely on the Applesoft version of VTAB, so it can be called even when the Applesoft image is switched out. I also draw a border around the screen image: a line of dashes above and below, and vertical lines up down both sides.

Instead of using $8D as a line length to turn off the screen output, I masked out the flag bit in $7F8+SLOT. This works in the Grappler and Grappler Plus interfaces, whereas the former method did not. (It is equivalent to printing control-I and letter-N.)

Further, I now restore the value of BASL,H at line 1490. Otherwise the value in CV ($25) and the address in BASL,H do not agree after printing the screen.

The last enhancement is at lines 1340-1370. Here I now convert characters from flashing and inverse modes to normal mode, or to blanks in some cases. You might want to arrange for a different mapping here, according to your own taste.

Even with all these enhancements, the program is still only 86 bytes long. The first version could be loaded anywhere without reassembly, because there are no internal references. The second version does have an internal JSR, so it would have to be reassembled to run at other locations, or modified to be made run-anywhere.

```
                    1000 *--------------------------------
                    1010 *       INSTANT HARDCOPY PROGRAM
                    1020 *       BY ULF SCHLICHTMANN
                    1030 *--------------------------------
0001-               1040 SLOT    .EQ 1
0028-               1050 BASL    .EQ $28
0029-               1060 BASH    .EQ $29
                    1070 *--------------------------------
0678-               1080 COLUMNS     .EQ $678
03EA-               1090 DOS.REHOOK  .EQ $03EA
F25A-               1100 AS.VTAB     .EQ $F25A
FE95-               1110 MON.PR      .EQ $FE95
FD8E-               1120 MON.CROUT   .EQ $FD8E
FDED-               1130 MON.COUT    .EQ $FDED
FE93-               1140 MON.SETVID  .EQ $FE93
                    1150 *--------------------------------
                    1160         .OR $300
0300- A9 01         1170 HCOPY   LDA #SLOT          SET UP OUTPUT VECTOR
0302- 20 95 FE      1180         JSR MON.PR         TO POINT AT PRINTER
0305- 20 8E FD      1190         JSR MON.CROUT      START A NEW LINE
0308- 8D 79 06      1200         STA COLUMNS+SLOT   DISABLE SCREEN
030B- A2 00         1210         LDX #0             START AT TOP OF SCREEN
030D- 20 5A F2      1220 .1      JSR AS.VTAB        COMPUTE BASE ADDRESS
0310- A0 00         1230         LDY #0             START IN COLUMN 1
0312- B1 28         1240 .2      LDA (BASL),Y       NEXT CHARACTER FROM THIS
0314- 20 ED FD      1250         JSR MON.COUT                              LINE
0317- C8            1260         INY
0318- C0 28         1270         CPY #40            END OF LINE YET?
031A- D0 F6         1280         BNE .2             NO
031C- 20 8E FD      1290         JSR MON.CROUT
031F- E8            1300         INX                NEXT LINE
0320- E0 18         1310         CPX #24            END OF SCREEN YET?
0322- D0 E9         1320         BNE .1             NO
0324- 20 93 FE      1330         JSR MON.SETVID
0327- 4C EA 03      1340         JMP DOS.REHOOK
```

```
                        1000 *  S.SCREEN PRINTER.PLUS
                        1010 *--------------------------------
                        1020 *       INSTANT HARDCOPY PROGRAM
                        1030 *       BY ULF SCHLICHTMANN
                        1040 *--------------------------------
0001-                   1050 SLOT    .EQ 1
0028-                   1060 BASL    .EQ $28
00FC-                   1070 VLINE   .EQ $FC
                        1080 *--------------------------------
07F8-                   1090 FLAGS       .EQ $7F8
03EA-                   1100 DOS.REHOOK  .EQ $03EA
FC22-                   1110 MON.VTAB    .EQ $FC22
FC24-                   1120 MON.VTABZ   .EQ $FC24
FE95-                   1130 MON.PR      .EQ $FE95
FD8E-                   1140 MON.CROUT   .EQ $FD8E
FDED-                   1150 MON.COUT    .EQ $FDED
FE93-                   1160 MON.SETVID  .EQ $FE93
FD9E-                   1170 MON.DASH    .EQ $FD9E
                        1180 *--------------------------------
                        1190         .OR $300
                        1200
0300- A9 01             1210 HCOPY   LDA #SLOT          SET UP OUTPUT VECTOR
0302- 20 95 FE          1220         JSR MON.PR         TO POINT AT PRINTER
0305- 20 8E FD          1230         JSR MON.CROUT      START A NEW LINE
0308- AD F9 07          1240         LDA FLAGS+SLOT
030B- 29 BF             1250         AND #$BF
030D- 8D F9 07          1260         STA FLAGS+SLOT
0310- 20 4B 03          1270         JSR DASH.LINE
0313- A2 00             1280         LDX #0             START AT TOP OF SCREEN
0315- 8A                1290 .1      TXA
0316- 20 24 FC          1300         JSR MON.VTABZ      COMPUTE BASE ADDRESS
0319- A9 FC             1310         LDA #VLINE
031B- 20 ED FD          1320         JSR MON.COUT
031E- A0 00             1330         LDY #0             START IN COLUMN 1
0320- B1 28             1340 .2      LDA (BASL),Y       NEXT CHARACTER
0322- 09 80             1350         ORA #$80           BE SURE IN RANGE
0324- C9 A0             1360         CMP #$A0
0326- B0 02             1370         BCS .3
0328- A9 A0             1380         LDA #$A0           PRINT SPACE IF ILLEGAL
032A- 20 ED FD          1390 .3      JSR MON.COUT
032D- C8                1400         INY
032E- C0 28             1410         CPY #40            END OF LINE YET?
0330- D0 EE             1420         BNE .2             NO
0332- A9 FC             1430         LDA #VLINE
0334- 20 ED FD          1440         JSR MON.COUT
0337- 20 8E FD          1450         JSR MON.CROUT
033A- E8                1460         INX                NEXT LINE
033B- E0 18             1470         CPX #24            END OF SCREEN YET?
033D- D0 D6             1480         BNE .1             NO
033F- 20 4B 03          1490         JSR DASH.LINE
0342- 20 22 FC          1500         JSR MON.VTAB       RE-ESTABLISH CURSOR
0345- 20 93 FE          1510         JSR MON.SETVID
0348- 4C EA 03          1520         JMP DOS.REHOOK
                        1530 *--------------------------------
                        1540 DASH.LINE
034B- A0 2A             1550         LDY #42
034D- 20 9E FD          1560 .1      JSR MON.DASH
0350- 88                1570         DEY
0351- D0 FA             1580         BNE .1
0353- 4C 8E FD          1590         JMP MON.CROUT
                        1600 *--------------------------------
```

Optional Patch for TEXT/ Command.....Bob Sander-Cederlof

Several have asked how to patch the character output at the beginning of each line by the TEXT/ command. TEXT/ normally writes your source code as a text file with control-I in place of each line number.

At $1AAD in the mother-board version, or $DAAD in the language card version, you will find $88. This is control-I minus one. Put what every character you wish there, less one. For example, if you want a leading space on each line, put $1F in $1AAD and/or $DAAD.

Division.....Bob Sander-Cederlof

Remembering long division in decimal can be hard enough, but visualizing it in binary and implementing it in 6502 assembly language is awesome! Study the following example, in which I divide an 8-bit value by a 4-bit value:

```
                 00110            6
               ----------        ---
      1101 ) 01010101      13 ) 85
  step A:    -0000             -78
             ----               --
              1010               7
  step B:     0000
              ----
              10101
  step C:     1101
              -----
              10000
  step D:     1101
              -----
               0111
  step E:     0000
              ----
               0111   Remainder
```

In the binary version, I have not made any leaps ahead like we do in decimal. That is, I wrote out the steps even when the quotient digit = 0. Now let's see a program which divides an 8-bit value by a 4-bit value, just like the example above.

```
                      1000 *  S.DIV.8.BY.4
                      1010 *--------------------------------
                      1020 *       DIVIDE 8-BIT VALUE
                      1030 *           BY 4-BIT VALUE
                      1040 *--------------------------------
0000-                 1050 DIVIDEND   .EQ 0
0001-                 1060 DIVISOR    .EQ 1
0002-                 1070 QUOTIENT   .EQ 2
                      1080 *--------------------------------
                      1090 S.DIV.8.BY.4
0800- A0 05           1100         LDY #5         COUNT OFF 5 STEPS
0802- A9 00           1110         LDA #0
0804- 85 02           1120         STA QUOTIENT
0806- A5 01           1130         LDA DIVISOR         SEE IF DIVISOR IN RANGE
0808- F0 1E           1140         BEQ .3              DIVIDE BY ZERO IS ILLEGAL
080A- 0A              1150         ASL            SHIFT DIVISOR TO LEFT NYBBLE
080B- 0A              1160         ASL
080C- 0A              1170         ASL
080D- 0A              1180         ASL
080E- 85 01           1190         STA DIVISOR
0810- A5 00           1200 .1      LDA DIVIDEND        COMPARE DIVIDEND TO DIVISOR
0812- 38              1210         SEC
0813- E5 01           1220         SBC DIVISOR
0815- 90 07           1230         BCC .2              DIVIDEND IS SMALLER
0817- C5 01           1240         CMP DIVISOR         SEE IF STILL LARGER
0819- B0 0D           1250         BCS .3              YES, OVERFLOW
081B- 38              1260         SEC                 SET QUOTIENT BIT   1
081C- 85 00           1270         STA DIVIDEND
081E- 26 02           1280 .2      ROL QUOTIENT        SHIFT QUOTIENT BIT IN
0820- 46 01           1290         LSR DIVISOR         SHIFT DIVISOR OVER
0822- 88              1300         DEY
0823- D0 EB           1310         BNE .1              DO NEXT STEP
0825- 26 01           1320         ROL DIVISOR    RESTORE DIVISOR
0827- 60              1330         RTS
0828- 00              1340 .3      BRK            DIVIDE FAULT
```

If you think this is a clumsy program, you may be right. Note that the loop runs five times, not four. This is because there are five steps, as you can see in the sample division above.

The first thing the program does is to clear the quotient value. In a 4-bit machine performing 8-bit by 4-bit division would yield a 4-bit quotient, so the top bits must be cleared. The rest of the bits will be shifted in as the division progresses.

Next the divisor is shifted up to the high nybble position, to align with the left nybble of the dividend. This is equivalent to step A in the example above. The loop running from line 1200 through line 1310 performs the five partial divisions.

If the divisor is zero, or if the first partial division proves that the quotient will not fit in four bits, the program branches to ".3". I put a BRK opcode there, but you would put an error message printer, or whatever.

To run the program above, I typed:

```
:$0:55 0D N 800G 0.2
```

and Apple responded with: 0000- 07 0D 06

which means the remainder is 7, and the quotient is 6.

Dividing Bigger Values:

The following program will divide one two-byte value by another. The program assumes that both the dividend and the divisor are positive values between 0 and 65535. This program was in the original Apple II monitor ROM at $FB84, but is not present in the Apple II Plus and Apple //e ROMs.

```
                        1000 *  S.DIV.16/16
                        1010 *---------------------------------
                        1020 *       DIVIDE 16 BY 16
                        1030 *---------------------------------
0050-                   1040 ACL     .EQ $50
0051-                   1050 ACH     .EQ $51
0052-                   1060 XTNDL   .EQ $52
0053-                   1070 XTNDH   .EQ $53
0054-                   1080 AUXL    .EQ $54
0055-                   1090 AUXH    .EQ $55
                        1100 *---------------------------------
0800- A0 10             1110 DIVMON  LDY #16        INDEX FOR 16 BITS
0802- 06 50             1120 .1      ASL ACL        DIVIDEND/2, CLEAR QUOTIENT BIT
0804- 26 51             1130         ROL ACH
0806- 26 52             1140         ROL XTNDL
0808- 26 53             1150         ROL XTNDH
080A- 38                1160         SEC
080B- A5 52             1170         LDA XTNDL      TRY SUBTRACTING DIVISOR
080D- E5 54             1180         SBC AUXL
080F- AA                1190         TAX
0810- A5 53             1200         LDA XTNDH
0812- E5 55             1210         SBC AUXH
0814- 90 06             1220         BCC .2         TOO SMALL, QBIT=0
0816- 86 52             1230         STX XTNDL      OKAY, STORE REMAINDER
0818- 85 53             1240         STA XTNDH
081A- E6 50             1250         INC ACL        SET QUOTIENT BIT = 1
081C- 88                1260 .2      DEY            NEXT STEP
081D- D0 E3             1270         BNE .1
081F- 60                1280         RTS
```

As written, this program expects the XTNDL and XTNDH bytes to be zero initially. If they are not, a 32-bit by 16-bit division is performed; however, there is no error checking for overflow or divide fault conditions.

This program builds the quotient in the same memory locations used for the dividend. As the dividend is shifted left to align with the divisor (opposite but equivalent to the shifting done in the previous program), empty bits appear on the right end of the dividend register. These bit positions can be filled with the quotient as it develops.

Signed Division

With a few steps of preparation, we can divide signed values using an unsigned division subroutine. All we need to remember is the rule learned in high school: If numerator and denominator have the same sign, the quotient is positive; if not, the quotient is negative.

```
                        1290 *--------------------------------
                        1300 *       SIGNED DIVISION 32/16
                        1310 *--------------------------------
002F-                   1320 SIGN    .EQ $2F
                        1330 *--------------------------------
                        1340 SIGNED.DIV.MON
0820- A0 00             1350         LDY #0
0822- 84 52             1360         STY XTNDL      CLEAR ACC EXTENSION
0824- 84 53             1370         STY XTNDH
0826- 84 2F             1380         STY SIGN
0828- A2 50             1390         LDX #ACL
082A- 20 3F 08          1400         JSR ABS
082D- A2 54             1410         LDX #AUXL
082F- 20 3F 08          1420         JSR ABS
0832- 20 00 08          1430         JSR DIVMON
0835- A5 2F             1440         LDA SIGN
0837- 10 05             1450         BPL .1         RESULT POSITIVE
0839- A2 50             1455         LDX #ACL
083B- 20 47 08          1460         JSR COMPLEMENT
083E- 60                1470 .1      RTS
                        1480 *--------------------------------
083F- B5 01             1490 ABS     LDA 1,X        LOOK AT SIGN
0841- 10 0F             1500         BPL ABSRET     POSITIVE
0843- 45 2F             1510         EOR SIGN       COMPLEMENT RESULT SIGN
0845- 85 2F             1520         STA SIGN
                        1530 COMPLEMENT
0847- 38                1540         SEC
0848- 98                1550         TYA            =0
0849- F5 00             1560         SBC 0,X
084B- 95 00             1570         STA 0,X
084D- 98                1580         TYA            =0
084E- F5 01             1590         SBC 1,X
0850- 95 01             1600         STA 1,X
0852- 60                1610 ABSRET  RTS
```

Double Precision, Almost:

What if I want to divide a full 32-bit value by a full 16-bit value? Both values are unsigned. The 32-bit dividend may have a value from 0 to 4294967295, and the divisor from 0 to 65535. All of the published programs I could find assume the leading bit of the dividend is zero, limiting the range to half of the above.

S-C Macro Cross Assemblers

The high cost of dedicated microprocessor development systems has forced many technical people to look for alternate methods to develop programs for the various popular microprocessors. Combining the versatile Apple II with the S-C Macro Assembler provides a cost effective and powerful development system. Hobbyists and engineers alike will find the friendly combination the easiest and best way to extend their skills to other microprocessors.

The S-C Macro Cross Assemblers are all identical in operation to the S-C Macro Assembler; only the language assembled is different. They are sold as upgrade packages to the S-C Macro Assembler. The S-C Macro Assembler, complete with 100-page reference manual, costs $80; once you have it, you may add as many Cross Assemblers as you wish at a nominal price. The following S-C Macro Cross Assembler versions are now available, or soon will be:

Motorola:	6800/6801/6802	now	$32.50
	6805	now	$32.50
	6809	now	$32.50
	68000	now	$50
Intel:	8048	now	$32.50
	8051	now	$32.50
	8085	soon	$32.50
Zilog:	Z-80	now	$32.50
RCA:	1802/1805	soon	$32.50
Rockwell:	65C02	now	$20

The S-C Macro Assembler family is well known for its ease-of-use and powerful features. Thousands of users in over 30 countries and in every type of industry attest to its speed, dependablility, and user-friendliness. There are 20 assembler directives to provide powerful macros, conditional assembly, and flexible data generation. INCLUDE and TARGET FILE capabilities allow source programs to be as large as your disk space. The integrated, co-resident source program editor provides global search and replace, move, and edit. The EDIT command has 15 sub-commands combined with global selection.

Each S-C Assembler diskette contains two complete ready-to-run assemblers: one is for execution in the mother-board RAM; the other executes in a 16K RAM Card. The HELLO program offers menu selection to load the version you desire. The disks may be copied using any standard Apple disk copy program, and copies of the assembler may be BSAVEd on your working disks.

S-C Software Corporation has frequently been commended for outstanding support: competent telephone help, a monthly (by subscription) newsletter, continuing enhancements, and excellent upgrade policies.

If the leading bit of the dividend is significant, a one bit extension is needed in the division loop. The following program implements a full 32/16 division.

```
                    1000 *  S.DIVIDE 32/16
                    1010 *--------------------------------
0800- A2 11         1020 DIVIDE LDX #17              16-BIT DIVISOR
0802- 18            1040        CLC                  START WITH NO OVERFLOW
0803- 6E 40 08      1050 .1     ROR OVERFLOW
0806- 38            1060        SEC
0807- AD 3B 08      1070        LDA DIVIDEND+1       NEXT-TO-HIGHEST BYTE
080A- ED 3F 08      1080        SBC DIVISOR+1        LEAST SIGNIFICANT BYTE
080D- A8            1090        TAY                  SAVE RESULT
080E- AD 3A 08      1100        LDA DIVIDEND         HIGHEST BYTE
0811- ED 3E 08      1110        SBC DIVISOR
0814- B0 05         1120        BCS .2               QUOTIENT BIT = 1
0816- 0E 40 08      1130        ASL OVERFLOW         TRUE QUOTIENT BIT
0819- 90 06         1140        BCC .3
081B- 8C 3B 08      1150 .2     STY DIVIDEND+1       QUOTIENT BIT = 1
081E- 8D 3A 08      1160        STA DIVIDEND
0821- 2E 3D 08      1170 .3     ROL DIVIDEND+3       SHIFT QUOTIENT BIT IN
0824- 2E 3C 08      1180        ROL DIVIDEND+2       AND MOVE TO NEXT POSITION
0827- 2E 3B 08      1190        ROL DIVIDEND+1
082A- 2E 3A 08      1200        ROL DIVIDEND
082D- CA            1210        DEX
082E- D0 D3         1220        BNE .1
0830- 6E 3A 08      1230        ROR DIVIDEND         SHIFT REMAINDER BACK IN
0833- 6E 3B 08      1240        ROR DIVIDEND+1
0836- 6E 40 08      1250        ROR OVERFLOW         SET SIGN BIT IF OVERFLOW
0839- 60            1260        RTS
                    1270 *--------------------------------
083A-               1280 DIVIDEND   .BS 4
083A-               1290 REMAINDER  .EQ DIVIDEND
083C-               1300 QUOTIENT   .EQ DIVIDEND+2
083E-               1310 DIVISOR    .BS 2
0840-               1320 OVERFLOW   .BS 1
                    1330 *--------------------------------
```

Line 1020 sets up a 17-step loop, because the 16-bit divisor can be shifted to 17 different positions under the 32-bit dividend. To make it easier to understand the layout of bytes in memory, I departed from the usual low-byte-first-format in this program. I assume this time that the most significant bytes are first:

```
Dividend:   $83A $83B $83C $83D
            msb . . . . . lsb

Divisor:    $83E $83F
            msb...lsb
```

I also have written this program to feed the quotient bits into the least significant end of the dividend register, as the dividend shifts left. The remainder will be found in the left two bytes of the dividend register, and the quotient in the right two bytes.

Watching It All Work:

Not being quite clairvoyant, I wanted to see what was really happening inside the 32/16 division program. So I added some trace printouts by inserting "JSR TRACE" right after lines 1050 and 1250. I also moved the variables into page zero, to show how much memory that can save. (All memory references are changed from 3-byte instructions to 2-byte instructions.)

```
                    1000 *  S.DIVIDE 32/16 WITH TRACE
                    1010 *--------------------------------
0000-               1020 OVERFLOW    .EQ $00
0001-               1030 DIVIDEND    .EQ $01 THRU $04
0001-               1040 REMAINDER   .EQ DIVIDEND
0003-               1050 QUOTIENT    .EQ DIVIDEND+2
0005-               1060 DIVISOR     .EQ $05 AND $06
                    1070 *--------------------------------
FD8E-               1080 MON.CROUT   .EQ $FD8E
FDDA-               1090 MON.PRHEX   .EQ $FDDA
FDED-               1100 MON.COUT    .EQ $FDED
                    1110 *--------------------------------
0800- A2 11         1120 DIVIDE LDX #17             16-BIT DIVISOR.
0802- 18            1130        CLC                 START WITH NO OVERFLOW
0803- 66 00         1140 .1     ROR OVERFLOW
0805- 20 2D 08      1150        JSR TRACE
0808- 38            1160        SEC
0809- A5 02         1170        LDA DIVIDEND+1      NEXT-TO-HIGHEST BYTE
080B- E5 06         1180        SBC DIVISOR+1       LEAST SIGNIFICANT BYTE
080D- A8            1190        TAY                 SAVE RESULT
080E- A5 01         1200        LDA DIVIDEND        HIGHEST BYTE
0810- E5 05         1210        SBC DIVISOR
0812- B0 04         1220        BCS .2              QUOTIENT BIT = 1
0814- 06 00         1230        ASL OVERFLOW        TRUE QUOTIENT BIT
0816- 90 04         1240        BCC .3
0818- 84 02         1250 .2     STY DIVIDEND+1      QUOTIENT BIT = 1
081A- 85 01         1260        STA DIVIDEND
081C- 26 04         1270 .3     ROL DIVIDEND+3      SHIFT QUOTIENT BIT IN
081E- 26 03         1280        ROL DIVIDEND+2      AND MOVE TO NEXT POSITION
0820- 26 02         1290        ROL DIVIDEND+1
0822- 26 01         1300        ROL DIVIDEND
0824- CA            1310        DEX
0825- D0 DC         1320        BNE .1
0827- 66 01         1330        ROR DIVIDEND        SHIFT REMAINDER BACK IN
0829- 66 02         1340        ROR DIVIDEND+1
082B- 66 00         1350        ROR OVERFLOW        SET SIGN BIT IF OVERFLOW
                    1360 *--------------------------------
082D- A9 B0         1370 TRACE  LDA #$B0
082F- 24 00         1380        BIT OVERFLOW
0831- 10 02         1390        BPL .1
0833- A9 B1         1400        LDA #$B1
0835- 20 ED FD      1410 .1     JSR MON.COUT
0838- A0 00         1420        LDY #0
083A- A9 A0         1430 .2     LDA #$A0
083C- 20 ED FD      1440        JSR MON.COUT
083F- B9 01 00      1450        LDA DIVIDEND,Y
0842- 20 DA FD      1460        JSR MON.PRHEX
0845- C8            1470        INY
0846- C0 04         1480        CPY #4
0848- 90 F0         1490        BCC .2
084A- 20 8E FD      1500        JSR MON.CROUT
084D- 60            1510        RTS
                    1520 *--------------------------------
```

The trace program prints first the overflow extension bit. If this is "1" on the last line, the quotient is too large to fit in 16-bits. TRACE next prints the four hex-digits of the quotient, and lastly the remainder. A line is printed before each step, and at the end to show the final results.

Now here are the printouts for a few values of dividend and divisor.

```
*1:00 00 FF FF  00 0A

*800G
0 00 00 FF FF       0 00 0B FE 19
0 00 01 FF FE       0 00 03 FC 33
0 00 03 FF FC       0 00 07 F8 66
0 00 07 FF F8       0 00 0F F0 CC
0 00 0F FF F0       0 00 0B E1 99
0 00 0B FF E1       0 00 03 C3 33
0 00 03 FF C3       0 00 07 86 66
0 00 07 FF 86       0 00 0F 0C CC
0 00 0F FF 0C       0 00 05 19 99 $FFFF / $0A = $1999 rem 5
```

```
*1:00 00 19 99  00 0A

*800G
0 00 00 19 99
0 00 00 33 32
0 00 00 66 64
0 00 00 CC C8
0 00 01 99 90
0 00 03 33 20
0 00 06 66 40
0 00 0C CC 80
0 00 05 99 01
0 00 0B 32 02
0 00 02 64 05
0 00 04 C8 0A
0 00 09 90 14
0 00 13 20 28
0 00 12 40 51
0 00 10 80 A3
0 00 0D 01 47
0 00 03 02 8F

$1999 / $0A = $28F rem 3
```

```
*1:FF F8 00 00  FF FF

*800G
0 FF F8 00 00
1 FF F0 00 00
1 FF E2 00 01
1 FF C6 00 03
1 FF 8E 00 07
1 FF 1E 00 0F
1 FE 3E 00 1F
1 FC 7E 00 3F
1 F8 FE 00 7F
1 F1 FE 00 FF
1 E3 FE 01 FF
1 C7 FE 03 FF
1 8F FE 07 FF
1 1F FE 0F FF
0 3F FE 1F FF
0 7F FC 3F FE
0 FF F8 7F FC
0 FF F8 FF F8

$FFF80000 / $FFFF = $FFF8 rem $FFF8
```

```
*1:FF FE 00 01  FF FF

*800G
0 FF FE 00 01
1 FF FC 00 02
1 FF FA 00 05
1 FF F6 00 0B
1 FF EE 00 17
1 FF DE 00 2F
1 FF BE 00 5F
1 FF 7E 00 BF
1 FE FE 01 7F
1 FD FE 02 FF
1 FB FE 05 FF
1 F7 FE 0B FF
1 EF FE 17 FF
1 DF FE 2F FF
1 BF FE 5F FF
1 7F FE BF FF
0 FF FF 7F FF
0 00 00 FF FF

$FFFE0001 / $FFFF = $FFFF
```

```
*1:FF FF FF FF  FF FF

*800G
0 FF FF FF FF
0 00 01 FF FF
0 00 03 FF FE
0 00 07 FF FC
0 00 0F FF F8
0 00 1F FF F0
0 00 3F FF E0
0 00 7F FF C0
0 00 FF FF 80
0 01 FF FF 00
0 03 FF FE 00
0 07 FF FC 00
0 0F FF F8 00
0 1F FF F0 00
0 3F FF E0 00
0 7F FF C0 00
0 FF FF 80 00
1 00 00 00 01

$FFFFFFFF / $FFFF = $0001 overflow
```

Short Note About Prime Benchmarks.....Frank Hirai
West Lebanon, NH

About your faster primes articles (Vol 2 #1, Vol 2 #5, and Vol 3 #2).... If you go back to Jim Gilbreath's original BYTE article you will find that the times he lists are for TEN iterations. As such they are not unreasonable for Integer BASIC and Applesoft. When comparing times for your 6502 assembly language versions, remember to multiply by ten!

Even so, 1.83 seconds for 10 iterations using Anthony Brightwell's program in the Apple compares quite well against 1.12 seconds for 10 iterations in an 8 MHz Motorola 68000.

[...and wait till we try it on a Number Nine 6502 card at 3.6 MHz! Or with a 65C02!]

Patching Applesoft for Garbage-Collection Indicator

.....Lee Meador

I wanted to know when (how often and for how long) Applesoft was doing garbage collection. The following patch will cause an inverse "!" to placed in the lower right hand corner of the screen whenever garbage collection takes place.

It is a little tricky to patch Applesoft, since it is in ROM! The first step is to copy the ROMs into the language card RAM space (any slot 0 RAM card will do). If you have an old Apple II with Integer BASIC on the mother board, you can do this by booting the DOS 3.3 Master. Otherwise, here are the steps:

```
]CALL-151
*C081 C081
*D000<D000.FFFFM
```

Next you need to place some code inside the Applesoft image in the RAM card. I chose to place the new code on top of the HFIND subroutine at $F5CB. (The code from $F5CB through $F5FF is never used by Applesoft.) Here is the routine I put there:

```
PATCH  PHA
       LDA #$21        INVERSE "!"
       STA $7F7        BOTTOM RIGHT CORNER
       PLA
       JSR GARBAG
       PHA
       LDA #$A0        BLANK BACK ON SCREEN CORNER
       STA $7F7
       PLA
       RTS
```

You also need to patch the existing "JSR GARBAG" inside Applesoft to jump to this new code. Here are the patches in hex:

```
*C083 C083          write enable RAM card
*E47B:CB F5
*F5CB:48 A9 21 8D F7
*F5D0:07 68 20 84 E4 48 A9 A0
*F5E0:8D F7 07 68 60
*C080               write protect RAM card
*control-C
]RUN your program
```

Here is a little Applesoft program which generates a lot of garbage strings so you can see the patch in action:

```
100 DIM A$(100)
110 FOR I = 1 TO 100
120 FOR J = 1 TO 200 : A$(I) = A$(I) + "B"   NEXT
130 PRINT I, : NEXT
```

Try running the program with different HIMEM values, to see the different effects.

S-C Macro Assembler, Version 1.1

A new version of the S-C Macro Assembler is just about ready, and it's going to be great!

I have added many new features, corrected a few problems, and created a special version to take advantage of the extra features of the new Apple //e computer. Here's a summary of the new items, so far:

New or Extended Features:

1. 80-column support! The release disk will now include versions for the Videx, STB, and maybe other 80-column cards.

2. The .HS directive now allows optional "." characters before and after each pair of hex digits. (e.g., .HS ..12..34..AB) This makes for easier counting of bytes, and allows you to put meaningful comments above or below the .HS lines.

3. .DO--.FIN can now be nested to 63 levels, rather than just 8 levels.

4. Comment lines may now begin with either "*" or ";".

5. Added .SE directive, which allows re-definable symbols. Now a macro can tell how many times it has been called.

6. Binary constants are now supported. The syntax is "%11000011101" (up to 16 bits).

7. ASCII literals with the high-bit set are now allowed, and are signified with the quotation mark: LDA #"X generates A9 D8. Note that a trailing "-mark is optional, just as is a trailing apostrophe with previous ASCII literals.

8. The TEXT/ <filename> command now outputs the current TAB character (default ctrl-I). It used to put out control-I no matter what the current TAB character was.

9. Now allow USER parameters to override memory protection. $101C-101D contains lower bound, and $101E-101F contains the upper bound of an area the user wants to UN-PROTECT. (The parameter for the starting page of the symbol table has moved from $101D to $1021, or $D01D to $D021.)

10. Added .PH and .EP directives, to start and end a phase. With these directives you can assemble a section of code that is intended to be moved and run somewhere else, without having to create a separate Target File.

11. Added .DUMMY and .ED to start and end a dummy section.

12. The TAB character may now be set to any character, including non-control characters, if you so desire.

...s to Known Problems:

1. .TI now properly spaces at top of each page, and at beginning of symbol table.

2. .AS and .AT now assemble lower case properly.

3. Changed the way the relative branches are assembled, so that "*" is equal to the location of the opcode byte. It used to be the location offset byte, which was non-standard.

4. Now pass two errors emit the proper number of object bytes, so that false range errors are not indicated.

Features added in support of Apple //e:

1. The Apple //e version allows you to change between 80- and 40- column screens at will, using PR#3 to go to 80-columns, or ESC-^Q to go to 40-columns.

2. In both normal input and edit modes, the DELETE key acts like a backspace key. It is interpreted the same as a left arrow (^H).

I haven't made up my mind yet about a new price, how we'll handle the upgrades, or how much the charge will be. We'll have the final details in next month's AAL.

More on the //e.....Bob Sander-Cederlof

1. Page Zero Usage:

Last month I erroneously reported that the new //e monitor used location $08 in page zero. It does not.

However, I was correct when I said the monitor now uses location $1F. It is possible that your programs conflict with this, and it is possible that some commercial programs conflict with this. For example, standard SWEET-16 uses $1F for half of its register 15, which is its PC-register.

If you disassemble the //e monitor at $FC9C (CLREOL, Clear to end of line), you will find a STY $1F a few lines down. This is the only visible place where $1F is used. However, there are some invisible ones lurking in the shadows of ROM.

2. The Shadow ROM:

By shadows, I mean the alternate ROM space which overlays the I/O slot ROMs. By switching the SLOTCX soft switch, the monitor turns on this shadow ROM; the rest of the code necessary in the new monitor is then accessible starting at $C100. At $FBB4 the new monitor saves the current status, disables interrupts and saves the status of the SLOTCX softswitch, and switches to the shadow ROM. Then it JMP's to $C100 with the Y-register indexing one of 9 or 10 functions.

The "shadow ROM" (my terminology, not Apple's) covers the address space from $C100-C2FF and $C400-C7FF. The space from $C300-$C3FF is also there, but it is always turned on in my //e. It holds the startup code for the 80-column card, and some memory management subroutines.

The space from $C100-C2FF contains the extra code for handling monitor functions in the //e. $C400-C7FF holds the self-test program that you initiate by pressing control-solid-apple-reset or control-both-apples-reset. (With both Apples, you get sound with the self-test.)

There is more ROM you switch in and out with another soft switch at $C800-CFFE. This holds the 80-column firmware.

3. Version ID Byte:

Location $FBB3 in the monitor identifies which type of Apple you have:

```
FBB3- 38 ... old Apple II
FBB3- EA ... Apple II Plus (Autostart Monitor)
FBB3- 06 ... Apple //e
```

This byte is now a permanent feature; Apple will continue to use it as an ID byte in the future. Art Schumer and Clif Howard published an extensive Version ID program in the February 1983 issue of Call APPLE. They listed two versions, one for use from DOS and one for use from Pascal.

Review: "The Visible Computer: 6502".....Bob Sander-Cederlof

For five years I have talked about it. "Someone should write a program that illustrates 6502 code being executed, using hi-res animation."

Software Masters never heard me, but they did it anyway! "The Visible Computer: 6502" is an animated simulation of our favorite microprocessor. You see inside the chip and watch the registers change, micro-step by micro-step. You even see the "hidden" registers: DL (data latch), DB (data buffer), IR (instruction register), and AD (address). You see HOW the instructions are executed.

I was amazed at the quality of the documentation. You get 140 pages of easy-to-follow, fun-to-read tutorial and reference text. The manual assumes only that you have an Apple, and are moderately familiar with Applesoft. It doesn't try to teach everything there is to know about machine language, but it does deliver the fundamental concepts.

Thirty demonstration programs are included on the disk, which progressively lead you through the instruction set. You begin with a two-byte register load, and work up to hi-res graphics and tone generation. All of the example programs are explained in detail in the manual. Of course, you can also trace your own programs or programs inside the Apple ROMs.

You can also use the simulator as a debugging tool, if your program will fit in the user memory area. The simulator provides a 1024-byte user memory, plus a simulated page zero and page one. You can also use $300-$3CF, if you wish. One unusual tool for debugging purposes is a full 4-function calculator mode, which works in binary, decimal, or hexadecimal.

Here is a list of the commands available at the normal level:

BASE	select binary, decimal, or hexadecimal
BLOAD	load a program to be simulated
BOOT	boot disk in slot 6, drive 1
CALC	turn on 4-function calculator
EDIT	short-cut entry of hex code into memory
ERASE	clear screen (so graphics can be seen)
L	disassemble five lines of code
LC	select memory for displayed in left column
PRINTER	turn on/off printer in slot one
RC	select memory for display in right column
RESTORE	restore normal screen display
STEP	select one of four simulation modes: 0 -- fastest, no display update until BRK 1 -- Full display, simulate until BRK 2 -- Full display, simulate one instruction with no pause between steps 3 -- Full display, simulate one instruction, pausing before each step

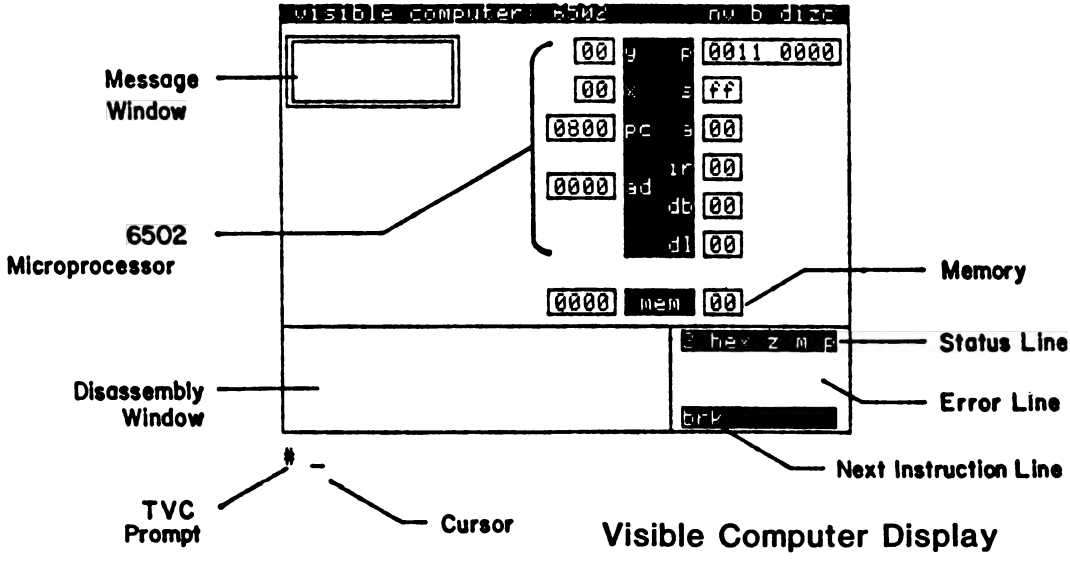
```
WINDOW          select one of three display options:
                MEM:  window shows 16 memory cells
                OPEN:  window is blank
                CLOSE:  window shows "hidden" 6502 registers
<addr><value>   store value at memory address
<reg><value>    store value in register
```

A "MASTER" mode can be turned on, which enables more features and commands for experienced users. In the master mode you can use the REAL zero page, you can modify any location in memory (even the ones that are dangerous!), you can BLOAD and BSAVE on standard DOS 3.3 disks, and run previously checked subroutines at full 6502 speed.

I know that a lot of you are looking for some help in understanding assembly language; "The Visible Computer" may be just the help you need. Let your own Apple teach you! Some of you are teaching 6502 classes; "The Visible Computer" is the most helpful teaching tools I have ever seen.

I was gratified to learn that the author is an old customer! He used an older version of the S-C Assembler for coding the longer examples, and the assembly language portions of the simulator. We even got a free plug on page 108!

The normal retail price of "The Visible Computer" is $49.95, our price will be an even $45 to readers of Apple Assembly Line.



Visible Computer Display

Apple Assembly Line is published monthly by S-C SOFTWARE CORPORATION, P.O. Box 280300, Dallas, Texas 75228. Phone (214) 324-2050. Subscription rate is $15 per year in the USA, sent Bulk Mail; add $3 for First Class postage in USA, Canada, and Mexico; add $13 postage for other countries. Back issues are available for $1.50 each (other countries add $1 per back issue for postage).